Dynamic Detection of Malicious Code in COTS Software

Martin Salois and Robert Charpentier
{Martin.Salois@drev.dnd.ca, Robert.Charpentier@drev.dnd.ca}

Defence Research Establishment Valcartier
2459 Pie XI Blvd. North
Val Bélair, Québec, Canada
G3J 1X5

April 2000

Abstract

COTS components are very attractive because they can substantially reduce development time and cost, but they pose significant security risks (e.g. backdoors, Trojan horses, time bombs, etc.).

These types of attack are not detected by standard virus detection utilities, which are essentially the only commercially available tools that work directly on binaries. This paper presents a dynamic approach that intends to address this problem.

The complexity of a real time-bomb attack that disables a program after a fixed period of time is shown. Building on this example, a method that works at the binary level and that could be used to facilitate the study of other time bombs — and hopefully of all types of malicious actions — is presented. This is the first step toward a fully automated tool to detect malicious actions in all their forms.

The method, which monitors processor instructions directly, is currently intended specifically for Windows NT running on an Intel processor. It could easily be extended to other platforms. This paper also discusses the possibility of using dynamic analysis techniques to overcome the inadequacy of the static methods.

Finally, a brief survey is presented of commercial tools that attempt to address this issue, considering where these products are today and what is needed to obtain a credible sense of security, as opposed to the often false sense offered by some commercial tools.

1 Introduction

COTS software has become the de facto standard in most organisations today. From management's point of view, it is often much more advantageous to buy certain products off-the-shelf than to develop them in-house. The final product is often cheaper both in time and in money. It is more robust and offers more features than what can be expected from in-house development, and it usually enjoys much better long-term support.

Unfortunately, an application that is developed by some other company — possibly in another country — can pose a serious security risk. Although the threat from viruses has been known for years and many potent commercial protection tools are available, other threats such as Trojan horses, time bombs, logic bombs, covert channels and so on are not as easily dealt with. Once they become known, virus detection tools can cover some of them, but the key is "become known": most detectors work only with known and already analysed threats. As will be shown, there are virtually no commercial tools offering a reasonable level of protection against unfamiliar attacks.

DREV initiated the MaliCOTS project in 1997 to address this situation. This paper first compares dynamic and static analysis techniques. Then some preliminary work on dynamic analysis is presented, focusing on time bombs. Last, a quick overview is given of some of the commercial tools currently available. A broader view of the project is presented in [1].

2 Static vs. Dynamic Analysis

Program analysis can be static or dynamic, or can use some other kind of technique that cannot clearly be classified as one or the other. This section takes a closer look at the advantages and disadvantages of static over dynamic analysis.

First, using static analysis allows malicious code to be detected without actually running the program; thus ensuring that any malicious actions discovered will never be executed. Also, static analysis can give a good idea of the program's behaviour, for all possible execution conditions. And there is no performance overhead associated with static analysis: after a single successful analysis, the program can run freely. But despite these beneficial properties, there are some inconveniences. The main drawback

Paper presented at the RTO IST Symposium on "Commercial Off-the-Shelf Products in Defence Applications "The Ruthless Pursuit of COTS"", held in Brussels, Belgium, 3-5 April 2000, and published in RTO MP-48.

to using static analysis is the undecidability of many interesting properties: they cannot be determined for all cases. Also, the analysed code needs not be the one that is actually run: changes can creep in between analysis and execution. The static analysis of source code is particularly vulnerable to this last difficulty, because the code must be compiled. Not only is there a possibility that a malevolent entity will modify the source code directly, but the language libraries used might be modified so that changes are not apparent.

Basically, dynamic analysis has the opposite pros and cons. One cannot detect malicious code dynamically before it is executed, give or take a few commands. For example, imagine a five-instruction sequence that, taken together, forms malicious code. An analysis tool might keep track of the last few instructions or use a list of suspicious instructions and be able to block the execution of the fifth command. However, this method could be rather limited on its own, because of the lack of a more global view. But dynamic analysis does not suffer from the undecidability characteristic of static analysis, because all run-time values are available or can be made available at any point in the program. Although dynamic analysis can have significant overhead in run-time performance, as compared to static analysis, in the end it has one major advantage: the analysed code is the code that actually runs, without any possibility of alteration.

Although some detection techniques cannot be clearly defined as static or dynamic, most are one or the other. Some innovative techniques, however, clearly use hybrid analysis. For example, Colby [3] proposes a way to define guards statically for loop expressions and to determine if they can be proven to be effective; if not, dynamic guards are inserted to be checked at run-time, when the boolean value of the expression can be computed.

It seems clear that static and dynamic techniques could very well be combined to ensure better success in the discovery of malicious code. A tool could do all that is possible with static analysis to identify vulnerable areas precisely and then use dynamic analysis to try to eliminate them. For example, the tool could pinpoint areas of code where it knows or can determine that static analysis will fail, and then concentrate on these segments using a dynamic method. Thus the overhead of a dynamic process running on top of a program could be greatly reduced, allowing better surveillance of an untrusted program without exceeding a tolerable level of intrusion.

3 Time Bomb Detection by Monitoring

Preliminary studies suggested the need to focus on a small subset of malicious code to begin with. Since a guinea pig was at hand — a time bomb in a program library that was being tested — it was decided to study time bombs more closely.

This section starts by giving a definition of a time bomb. Then the details of a time bomb case study are presented. Finally, all possible ways of getting the time in the Win32 subsystem are examined to outline a possible way to detect time bombs.

3.1 Definition

A *time bomb* is malicious code that is triggered in a program when a specific logical condition relating to time is met. "Time" here refers to the actual system time and date or a countdown in seconds, hours, days, or even months or years. Although it could be argued that limiting the number of executions (before declaring the expiration of evaluation software, for example) could be called a time bomb, in this analysis it is considered a logic bomb.

For the purposes of security and detection, it does not matter whether or not the time bomb was inserted intentionally. An unintentional time bomb can still compromise the system.

Typical examples of time bombs are time computations that prevent a program from working after x hours, minutes, days, etc. If this type of time bomb is used appropriately, perhaps to protect proprietary software, it is not really "malicious." Even so, the process is rarely done in a correct and standard way, as will be seen in the next section, so it is still considered an unacceptable risk.

Other time bombs include viruses that are launched at specific dates: one that wishes "Merry Christmas!" or that commemorates a special day.

Of course, many programs legitimately need to use time triggers. For example:

- Virus scanners that you can schedule to work every day at a specific time,
- Meeting schedulers that notify you of appointments,
- Automatic backup programs,
- Games that limit the time to finish a puzzle.

Deciding if a "time bomb" is malicious or not has been left out of present concerns, although possible ways of deciding automatically will be discussed later on. For the moment, our only interest is in locating them; automatic classification mechanisms are useless without tools for detection.

3.2 A Case Study

To get a feel of what a time bomb may look like in assembly code, along with its possibly great complexity, a real-world example — from a source that shall remain nameless — will be examined. It will simply be called *SoftBomb*.

SoftBomb is distributed as a DLL. A demo version is available that will only work for one month. Thus, in effect, there is a time bomb that detects that the same day in the next month has passed and triggers the stopping of the

Code Excerpt 1: A legitimate time bomb

```
//...
SYSTEMTIME systemTime;
GetSystemTime(&systemTime);
if( systemTime.wMonth > previousSystemTime.wMonth ){
    // Then do something
//...
```

executable. Since *SoftBomb* is a DLL, it cannot actually stop the execution; it sends an error message saying that the evaluation time has expired when you try to initialise it. For the sake of simplicity, we will continue to say that it "stops" execution.

A time bomb needs to get the system time from somewhere. As will be seen in the next subsection, there are many ways of doing this, even restricting our studies to legal Win32 methods. After obtaining the date/time, the time bomb will check it against an installation date that it stored somewhere safe — preferably a place unknown to the user so he cannot simply delete it. This last requirement is not actually part of the time bomb itself, so this paper will not explore the assembly details of how *SoftBomb* stores the installation date, only the general scheme and how it can be bad for user systems.

The idea behind a successful protection scheme is to make it as obscure and as irrational as possible. If it is done in the simplest and most sensible way, crackers will have an easy time breaking it. For example, say you want to know if the month has changed in a legitimate time bomb. You would simply proceed as is shown in Code Excerpt 1. However, this way of doing things would be too simple a protection scheme. As will be seen, *SoftBomb* is much more "clever."

Note that in Win32 systems, the time and the date travel around in the same structures most of the time. For instance, `GetSystemTime` gives both time and date. There is no function called `GetSystemDate`. Unless noted otherwise, the term time will be used to mean both time and date.

This subsection takes a look at how *SoftBomb* gets the system date, how it does multiple checks on it, and where the installation date is stored. As a bonus, for completeness and possible future use, a few pointers are given on how one might crack *SoftBomb*.

First it must be mentioned that the time bomb in *SoftBomb* is located in a particular function that the user must call to initialise the library before use. Since *SoftBomb* is a DLL, this simplifies the analysis a little because DLLs are meant to be used by other programs that are not supposed to know their inner workings. This means that they have clear-text names and clear-cut boundaries for functions. This fact allowed us to narrow the search to a small fraction of the whole DLL. This will not always be the case: a time bomb could be scattered over a much larger fragment of code.

3.2.1 Getting the Date

Since simply getting the date with the `GetSystemTime` function would be too obvious, *SoftBomb* uses another approach: it opens a file that it is certain to find in the main Windows NT directory, and gets the time of the last access to this file.

In this case, the file is `win.ini`. Since it is an essential configuration file for Windows NT, it is always present and, by opening it, *SoftBomb* updates its access time. Hence, it gets the system date after transforming it from a file-time structure to a system-time structure.

The actual code is shown in Code Excerpt 2 (note that all the assembly code in this section was obtained by using Sang Cho's powerful Windows Disassembler [2]). The comments after semicolons are inserted automatically by the disassembler, which identifies common Win32 API, even with some form of static def/use analysis as, for example, at line 29 where it knows that `ebp` contains the address of the function `CreateFile`, inserted at line 23. The comments in italics after two slashes were inserted manually after analysis, to ease comprehension and to explain what is going on in the lines that were skipped to save space. Reserved keywords for instructions are highlighted in bold. Line numbers are used because they are more convenient than memory addresses.

Looking at the code more closely, it can be seen on the first line that *SoftBomb* gets the system directory, which is `c:\winnt\system32`, and stores the address of this string in `ecx` (line 2), then in `eax` in the function called at line 5.

After that, it surreptitiously changes the string, character by character, until it becomes `c:\winnt\win.ini` (lines 9, 11, 16, 17, 18, 23, 25, 26), interlacing this change with the normal operations necessary for the next function call to the API `CreateFileA` function (line 28), which requires 7 parameters (the 7 preceding pushes). This function is used to open the existing file `win.ini`.

Then *SoftBomb* uses the functions `GetFileTime` and `FileTimeToSystemTime` to get the file's last access time and convert it into the desired system time format. *SoftBomb* now has the current system time and date to do with as it pleases.

3.2.2 Checking the Date

To be certain that a cracker could not simply change one jump instruction to crack it, *SoftBomb* checks the date two different ways. And then to be really sure, it checks again.

Code Excerpt 2: Getting the current date

```
1  :1F0017AD call ebp ;;jmp KERNEL32.GetSystemDirectoryA
2  :1F0017AF lea ecx, dword[esp+00000288] // "c:"winnt"system32" at address ecx
3  :1F0017B6 push 0000005C
4  :1F0017B8 push ecx
5  :1F0017B9 call 1F0056F0 // among other things, copies ecx in eax
6  :1F0017BE add esp, 00000008
7  :1F0017C1 test eax, eax
8  :1F0017C3 je 1F00195D
9  :1F0017C9 mov byte[eax+01], 57 // changes memory to "c:"winnt"Wystem32"
10 // ... The next 6 instructions are inc eax 6 times (so eax=eax+6).
11 :1F0017D3 mov byte[eax-04], 49 // changes memory to "c:"winnt"WIstem32"
12 :1F0017D7 inc eax
13 :1F0017D8 push 00000000 // hTemplateFile = NULL
14 :1F0017DA push 00000080 // dwFlagsAndAttributes = FILE_ATTRIBUTE_NORMAL
15 :1F0017DF push 00000003 // dwCreationDistribution = OPEN_EXISTING
16 :1F0017E1 mov byte[eax-04], 4E // changes memory to "c:"winnt"WINtem32"
17 :1F0017E5 mov byte[eax-03], 2E // changes memory to "c:"winnt"WIN.em32"
18 :1F0017E9 mov byte[eax-02], 49 // changes memory to "c:"winnt"WIN.Im32"
19 :1F0017ED lea ecx, dword[esp+00000294]
20 :1F0017F4 push 00000000 // lpSecurityAttributes = NULL
21 :1F0017F6 push 00000003 // dwShareMode
22 :1F0017F8 mov ebp, dword[1F013194]
23 :1F0017FE mov byte[eax-01], 4E // changes memory to "c:"winnt"WIN.IN32"
24 :1F001802 push 80000000 // dwDesiredAccess = GENERIC_READ
25 :1F001807 mov byte[eax], 49 // changes memory to "c:"winnt"WIN.INI2"
26 :1F00180A mov byte[eax+01], 00 // changes memory to "c:"winnt"WIN.INI"
27 :1F00180E push ecx // lpFileName = "C:"winnt"WIN.INI"
28 :1F00180F call ebp ;;jmp KERNEL32.CreateFileA // (7 parameters=7 pushes)
29 // ... Checks for errors. Loads the time in registers for the following pushes.
30 :1F001831 push eax // lpLastWriteTime
31 :1F001832 push ecx // lpLastAccessTime
32 :1F001833 push edx // lpCreationTime
33 :1F001834 push esi // hFile
34 :1F001835 call dword[1F013190] ;;jmp KERNEL32.GetFileTime // (4 params=4 pushes)
35 // ... Checks for errors. Closes the file. Puts the file time in registers for following pushes.
36 :1F00185E push eax // lpSystemTime
37 :1F00185F push esi // *lpFileTime
38 :1F001860 call edi ;;jmp KERNEL32.FileTimeToSystemTime // (2 params=2 pushes)
39 // ... Checks for errors.
```

So, there are three different checkpoint that are performed one after the other, each using different logic to see if the expiration year, date, and day have been reached or to check if the date itself has been tampered with. Let us look at them more closely.

The first checkpoint is pretty simple. It is shown in Code Excerpt 3. In the first line the installation date is compared with the current date, as obtained in the previous subsection. If the installation year is higher than the current year, it assumes there is an error, reset its structures and checks again. If there is still an error, it goes to the second checkpoint. There is actually a bug in *SoftBomb* at line 3: if the year has changed, it stops initialising *SoftBomb*, giving an expiration message. Therefore, if you install *SoftBomb* on December 31st, 1998, it expires on January 1st, 1999! Otherwise, the check continues by verifying that the expiration month has not passed. If it has, it ends the execution.

The second and third checkpoints are somewhat less independent than the first. In fact, they could probably be considered as a single checkpoint since the second jumps into the third to complete some checks. However, for clarity, it is better to view them as two different phases.

In Code Excerpt 4, the second checkpoint first compares the install year with the current year (line 1). If they are not the same, it moves on to the third checkpoint (line 2). If they are equal it checks the install month against the current month (line 4) and, if they are not the same, moves on to verify that only one month has passed and that the same day in the next month has not yet passed (line 7 and lines 10-19). If it is still the same month, it checks to see if the date is correct — that the system has not gone back in time — (line 7), decides that *SoftBomb* has not expired, and finishes its initialisation (jump at line 8).

SoftBomb then enters its third and final checkpoint, shown in Code Excerpt 5. At this point, it knows that the

Code Excerpt 3: First checkpoint

```
1  :1F00186E cmp word[esp+16], ax  // installation year, current year
2  :1F001873 ja 1F001897  // install year ¿ current year (error, double check)
3  :1F001875 jne 1F001A20  // stops if year has changed
4  :1F00187B mov eax, dword[esp+1A]
5  :1F00187F xor ecx, ecx
6  :1F001881 mov cx, word[esp+0000008A]
7  :1F001889 and eax, 0000FFFF
8  :1F00188E inc eax
9  :1F00188F cmp eax, ecx  // install month+1 (expiration month), current month
10 :1F001891 jl 1F001A20  // stops if expiration month ¡ current month
```

Code Excerpt 4: Second checkpoint

```
1  :1F001983 cmp word[esp+16], ax  // install year, current year
2  :1F001988 jne 1F0019C6  // if install != current, go to next checkpoint
3  :1F00198A mov ax, word[esp+26]
4  :1F00198F cmp word[esp+1A], ax  // install month, current month
5  :1F001994 jne 1F0019A2  // if install != current, check day
6  :1F001996 mov ax, word[esp+2A]
7  :1F00199B cmp word[esp+18], ax  // install day, current day
8  :1F0019A0 jbe 1F0019FC  // same year & same month & install current, OK
9  ---------  // Inserted by disassembler to indicate a block's ending/starting.
10 :1F0019A2 xor eax, eax
11 :1F0019A4 mov ecx, dword[esp+1A]
12 :1F0019A8 mov ax, word[esp+26]
13 :1F0019AD and ecx, 0000FFFF
14 :1F0019B3 sub eax, ecx  // current month, install month
15 :1F0019B5 cmp eax, 00000001
16 :1F0019B8 jne 1F0019C6  // if difference != 1, then go to next checkpoint
17 :1F0019BA mov ax, word[esp+2A]  // difference = 1, check if same day not reached
18 :1F0019BF cmp word[esp+18], ax  // install, current
19 :1F0019C4 jae 1f0019fc  // if same day next month not passed, ok
```

year has changed (actually, because of the previously mentioned bug, *SoftBomb* never gets here, but let us pretend it does). At lines 1, 2, and 3, it checks to see if the difference is only one year; if not, it stops. If the year difference is indeed only one, it moves on to check if the current month is January (lines 4 and 5) and if the current month is December (lines 6 and 7), the only possible situation for a one-month evaluation. If this is not the case, execution stops; if it is, one final verification is made to check that the expiration day has not passed. Finally, if all is clear, the program continues with its normal initialisation.

3.2.3 Storing the Date

In the previous subsection, the installation date was mentioned. But where does *SoftBomb* store the date on which it was installed? As already stated, this question is actually outside the scope of time-bomb detection. But the reader might be interested, so *SoftBomb*'s approach will be outlined here.

It was also previously mentioned that it can hardly be considered malicious for a company to try to protect its software but that the methods sometimes used can be quite malicious if they are not regular and standard. The following discussion supports this point.

Once again, when it comes to hiding information for a protection scheme, obscurity is the way to go. You want to hide the information as deeply as possible, in a place where the user will not look; or should he decide to look, where he will not find anything suspicious.

SoftBomb's protection scheme is cunning in this sense because it does nothing for the first few uses. It waits a random number of times before storing an installation date on the hard drive. And a careful or suspicious user monitoring the first few runs of *SoftBomb* to see if it is legitimate is unlikely to catch the suspicious write to the Registry — the Registry is where all of Windows NT's configurations are stored — because *SoftBomb* stores the installation date there using a key inconspicuously named `FontAttributes`. A key with this name would easily be overlooked, especially since it is placed in a region of the Registry where the configuration of the desktop is kept (registry path `HKEY_CURRENT_USER\Control Panel\desktop`). Among the legitimate keys stored at this place, there are `AutoEndTasks`, `Pattern`, `IconHorizontalSpacing`, `IconVerticalSpacing`, `TileWallPaper`, `Wallpaper`, and so on. It is easy to see why one called `FontAttributes` would not be looked at twice.

Code Excerpt 5: Third checkpoint

```
1  :1F0019D9 sub eax, ecx // install year, current year
2  :1F0019DB cmp eax, 00000001
3  :1F0019DE jne 1F001A20 // if difference != 1 then stop
4  :1F0019E0 cmp word[esp+26], 0001 // current month, January (01)
5  :1F0019E6 jne 1F001A20 // if current month not January then stop
6  :1F0019E8 cmp word[esp+1a], 000C // install month, December (c=12)
7  :1F0019EE jne 1f001a20 // if install month not December then stop
8  :1F0019F0 mov ax, word[esp+2a]
9  :1F0019F5 cmp word[esp+18], ax // install day, current day
10 :1F0019FA jc 1F001A20 // (jc=jb) if install day passed then stop
11 // ... Continue with normal initialization.
```

Finally, another random number of executions after expiration, *SoftBomb* creates another Registry key in the same registry path, called `DragDelay`. The purpose of this key is not completely clear, but it seems to be a flag that indicates that *SoftBomb* has expired. Since it is not really part of the time bomb itself, it was not investigated further.

Now that it has been shown how a real-world software product hides the installation date, it is trivial to demonstrate how the activity could be bad for a system: if programs were to write to the Registry anywhere they please, without ever cleaning up behind them, a maintenance nightmare would result. Legitimate and correct programs have a difficult enough task cleaning up their own mess; we cannot have programs writing where they are not supposed to. Clearly, such behaviour is unacceptable.

3.2.4 Cracking It

Only one matter remains to conclude this case study: how could the time bomb in *SoftBomb* be circumvented? Although some might perceive such action as a bad thing — after all, cracking software products is probably illegal in most countries — this example is only an illustration. The results of the work could later be extended to protect a system against more serious threats. For instance, a virus could be stopped dead in its tracks simply by dynamically stopping the time bomb that triggers it.

So, how could a "cracker" crack *SoftBomb*? That is, how can one remove the protection? There are many possible solutions. The two most plausible ones are given here or, at least, the two more practical in our view:

- Systematically replace all instructions that jump to the end sequence with `noops` in order to avoid ever getting to the stopping code. This could be done statically with a hexadecimal editor, or dynamically, on the fly.

- Add a routine to *SoftBomb* that would execute at the beginning of the initialisation function. This "hook" would simply delete the two registry keys identified in the previous subsection and transfer control back to the normal flow of the function.

Either solution could be used by a dynamic protection tool to thwart the time bomb, but the first seems more direct and easier to implement. One must simply reverse the jumps at run-time, a simple enough task for anyone familiar with debuggers.

This concludes our case study. The following sections look at the various ways to get the system time and date.

3.3 How to Get the Time and Date

This section explores the many ways to get the system time and date in Windows NT via the Win32 subsystem. The authors do not pretend that the list is exhaustive: smart, malicious attackers will always come up with new approaches. Also, it would take many pages to illustrate all the possible ways that the research team was able to devise to get the system time. A simple list of the Win32 functions that can provide the time or date and of the functions that can modify or control the time or date in any way. There are many of them, with many parameters that control their behaviour, and many functions that perform essentially the same task have different implementations with different names: `CreateFile`, `CreateFileA`, and `CreateFileEx`, for example. Consider this a first step in the construction of a database of knowledge on malicious code, a subject we will return to.

First, let us consider functions that can give the time of day (or the date) directly or indirectly in combination. Code Excerpt 6 presents the signatures of these functions. The function names are in boldface to make it easier to spot them among the parameters; they are presented in alphabetical order.

For historical reasons, many formats for the date/time exist and are still available. For example, a date can be computed from a long integer containing the number of seconds since 1970, or it can be directly stored as `dd-mm-yyyy`, or `yyyy-mm-dd`, and so on. This explains in part the large number of functions that can give the time/date.

It has been seen that the combination of `CreateFile`, `GetFileTime`, and `FileTimeToSystemTime` can be used to find the date. Following the same pattern, one could, for instance, create a file in MS-DOS mode (which is provided for backward compatibility), do

Code Excerpt 6: Functions that can be used to get and compare time and/or date

```
1  LONG CompareFileTime(CONST FILETIME *lpFileTime1, CONST FILETIME *lpFileTime2);
2  HANDLE CreateFile(LPCTSTR lpFileName, DWORD dwDesiredAccess, DWORD dwShareMode,
3     LPSECURITY_ATTRIBUTES lpSecurityAttributes, DWORD dwCreationDistribution,
4     DWORD dwFlagsAndAttributes, HANDLE hTemplateFile);
5  BOOL DosDateTimeToFileTime(WORD wFatDate, WORD wFatTime, LPFILETIME lpFileTime);
6  BOOL FileTimeToDosDateTime(CONST FILETIME *lpFileTime, LPWORD lpFatDate, LPWORD lpFatTime);
7  BOOL FileTimeToLocalFileTime(CONST FILETIME *lpFileTime, LPFILETIME lpLocalFileTime);
8  BOOL FileTimeToSystemTime(CONST FILETIME *lpFileTime, LPSYSTEMTIME lpSystemTime);
9  HANDLE FindFirstFile(LPCTSTR lpFileName, LPWIN32_FIND_DATA lpFindFileData);
10 HANDLE FindFirstFileEx(LPCTSTR lpFileName, FINDEX_INFO_LEVELS fInfoLevelId,
11    LPVOID lpFindFileData, FINDEX_SEARCH_OPS fSearchOp, LPVOID lpSearchFilter,
12    DWORD dwAdditionalFlags);
13 BOOL FindNextFile(HANDLE hFindFile, LPWIN32_FIND_DATA lpFindFileData);
14 BOOL GetFileTime(HANDLE hFile, LPFILETIME lpCreationTime, LPFILETIME lpLastAccessTime,
15    LPFILETIME lpLastWriteTime);
16 VOID GetLocalTime(LPSYSTEMTIME lpSystemTime);
17 LONG GetMessageTime(VOID);
18 VOID GetSystemTime(LPSYSTEMTIME lpSystemTime);
19 VOID GetSystemTimeAsFileTime(LPFILETIME lpSystemTimeAsFileTime);
20 DWORD GetTickCount(VOID);
21 NET_API_STATUS NetRemoteTOD(LPTSTR UncServerName, LPBYTE *BufferPtr);
22 LONG RegEnumKeyEx(HKEY hKey, DWORD dwIndex, LPTSTR lpName, LPDWORD lpcbName,
23    LPDWORD lpReserved, LPTSTR lpClass, LPDWORD lpcbClass, PFILETIME lpftLastWriteTime);
24 LONG RegQueryInfoKey (HKEY hKey, LPTSTR lpClass, LPDWORD lpcbClass, LPDWORD lpReserved,
25    LPDWORD lpcSubKeys, LPDWORD lpcbMaxSubKeyLen, LPDWORD lpcbMaxClassLen,
26    LPDWORD lpcValues, LPDWORD lpcbMaxValueNameLen, LPDWORD lpcbMaxValueLen,
27    LPDWORD lpcbSecurityDescriptor, PFILETIME lpftLastWriteTime);
28 BOOL ReportEvent(HANDLE hEventLog, WORD wType, WORD wCategory, DWORD dwEventID,
29    PSID lpUserSid, WORD wNumStrings, DWORD dwDataSize, LPCTSTR *lpStrings,
30    LPVOID lpRawData);
31 BOOL SystemTimeToFileTime(CONST SYSTEMTIME *lpSystemTime, LPFILETIME lpFileTime);
32 BOOL SystemTimeToTzSpecificLocalTime(LPTIME_ZONE_INFORMATION lpTimeZoneInformation,
33    LPSYSTEMTIME lpUniversalTime, LPSYSTEMTIME lpLocalTime);
34 MMRESULT timeGetSystemTime(LPMMTIME pmmt, UINT cbmmt);
35 DWORD timeGetTime(VOID);
```

a `GetFileTime`, and then do a `FileTimeToDosDateTime` to get the date in a different format that could be converted to system time.

Remember that the idea for a malicious scheme is to confuse an eventual detection process. So instead of creating a file, one could write a null character to a known file or simply open it. We will not attempt to cover all such variations.

The functions in lines 9, 10, and 13 could be used to go through the system directory files to extract the most recent date. Since the files in this directory are accessed often, at least the date will almost certainly be correct, if not the time. Similarly, the registry functions (lines 22, 24) could be used to get the last access time of often-used registry keys.

Via the logging mechanisms, it is possible to know when Windows NT was started. In many installations, the machines are rebooted every day. If that is case, the date is available directly. If it is not the case, functions that give the elapsed time since the last reboot (lines 20, 34) could be used to calculate the current date and time. A program could send a message to itself and then get the date of the event, which is automatically recorded by the mechanism. Many Win32 executables handle incoming messages — mouse clicks and keyboard commands, for instance — this way, so an attacker could use the function `GetMessageTime` to get the elapsed time between the starting of Windows NT and the handling of the message.

If an attacker knows that his target system obtains time information from a network, he can use the "network remote time of day" function (line 21).

This concludes our survey of how to get the time and date. Now, let us look at two ways to set a time bomb that do not require the application itself to look at the time. The signatures for these functions are given in Code Excerpt 7.

If the target is on a network, one can simply ask the system to wake the executable code up at a given future time. Of course, one must assume that it will still be running then.

Similarly, in the next two functions if one knows that the system runs for extended periods of time, one can set up a timer that will "beep" at regular intervals: several days or even weeks.

Code Excerpt 7: Functions to set the system time, a file time or to set a timer

```
1 NET_API_STATUS NetScheduleJobAdd(LPWSTR Servername, LPBYTE Buffer, LPDWORD JobId);
2 UINT SetTimer(HWND hWnd, UINT nIDEvent, UINT uElapse, TIMERPROC lpTimerFunc);
3 BOOL SetWaitableTimer(HANDLE hTimer, LARGE_INTEGER *pDueTime, LONG lPeriod,
4    PTIMERAPCROUTINE pfnCompletionRoutine, LPVOID lpArgToCompletionRoutine, BOOL fResume);
```

3.4 Monitoring for Time Bombs

Based on these examples, it is now possible to propose ways to detect a time bomb in an executable code. Let us look briefly at two possibilities:

- Hook — that is, "intercept and redirect" in Win32 terminology — all of the time-supplying functions that were enumerated in the previous section. Determine who calls them and watch the callers for anomalous behaviour.

- The detection tool itself can get the date and verify on the fly, at assembly instruction level, if any data that is equivalent to the date is used to determine the results of conditional jumps.

Once more, "time" here really means "time and date."

In the next two subsections, the pros and cons of these two semi-automatic approaches are explored, then a combination of the two is proposed for maximum benefit. The subsection concludes with possible ways to automate the process by the use of specifications.

3.4.1 Hooking the Time Functions

This approach requires a program to intercept all calls made to the functions enumerated in Subsection 3.3. Commercial and freeware programs that do this have been noted, so the task should not pose too great a technical difficulty.

This technique would be used in a certifying environment; i.e., a closed and clean environment in which to perform extensive tests on the target program. During these tests, if the executable calls a "time" function, a flag is raised to look more carefully at the program to see if its behaviour has changed from normal. If it has, the tool can pinpoint the region of code where the time was accessed from and, hopefully, indicate if there is indeed a time bomb at that point in the assembly code of the executable.

By itself, this method cannot actually stop a time bomb from being triggered; it can only indicate the possibility of triggering and narrow the region for a human search. However, the intrusion level is minimal and the method would not limit the number of tests that can be run.

Evidently, if an attacker can devise a way to access the system time and date that was not included, this method would be powerless to detect it.

3.4.2 Comparing with Current Time

In this method, a monitoring tool would be created that is similar to what Jeffery proposed in [5]. A full-blown virtual machine is not needed; only a way to control the execution of applications and the ability to examine (and possibly change) the target program's memory.

A specialised monitor is needed, one that gets the time and date for itself. Then it runs the target program, opcode by opcode, and checks to see if it uses data equivalent to the time or date to control the execution flow. If so, and if the tool is being used in a certifying environment, it raises a flag telling the test engineer where to check the code more carefully. If it is not being used in such an environment, all it could do is to stop the application at that point, warn the user, and wait for further instructions. Because assembler code could not be provided for the user to verify, the message would have to be much simpler.

This method is certainly more powerful than the preceding one because it includes it. Effectively, if the target program uses the time data after returning from one of the "time" functions, this method will catch it. This method is also much more intrusive than the other, and consequently would be much slower.

3.4.3 Combining the two

To thwart the second method, an ingenious attacker could simply add a fixed number to the day, month, and year. If he adds 10, for example, and the monitoring application knows that the date is "03-04-2000," it would not detect control-flow jumps that check against 13, 14, and 2010 respectively. It would think they are simply numbers that the target program uses for its normal procedures. This was illustrated in the first checkpoint of *SoftBomb* example (Code Excerpt 3), where *SoftBomb* adds one to the installation month to know the expiration month. It could as easily have subtracted one from the current month to achieve the same result.

In order to prevent this simple scheme from defeating the second technique, it should be combined with the "hook" technique. Statically, it can recognise a call to a precise API function. It would be a simple matter to stop the target program only on "time" functions, and start examining the application closely only from there. This would considerably reduce the level of intrusion. A simple form of dynamic def/use graph could also be implemented to keep track of the time data to determine if a control flow condition is using some modified form of it.

To sum things up, a good way to detect a time bomb dynamically would be:

1. Create a monitor that can:
   - control the execution of a target program,
   - break on any instruction, and
   - examine the content of its memory address space.
2. Determine statically where the "time" functions are called and insert breakpoints at these points.
3. Execute the target program step-by-step, keeping track of time data and checking to see if the flow of control is influenced by it. If so, raise a warning.

The first step poses only technical difficulties, depending on the machine, the operating system and its architecture. The second step is even simpler since a good disassembler, such as the one that was used in Subsection 3.2, will do most of the job for us.

The last step is not that complex either. It only requires a good def/use mechanism to keep track of variables. This is easily done for registers, but problems may arise when memory is used to store variables and data structures. A resourceful attacker could use quite complex data structures, including recursive ones, or could even encrypt the time data. Nonetheless, building a def/use graph dynamically is a lot easier than doing it statically. The only major problem that can be foreseen is the amount of memory required to keep a "virtual double" of all time-related variables.

So far only a semi-automatic tool has been discussed: the first logical step toward a fully automated tool. First, knowledge needs to be gathered and a great deal of experimentation on the subject is required to augment our experience before our team can even think of automating the process. Still, if an automated tool is ever to see the light of day, it is necessary to tell the tool what is and what is not expected from a program. The following subsection addresses this subject.

Of course, static analysis could be combined with a dynamic tool. In the MaliCOTS project, static analysis techniques to detect malicious code are under investigation. The current plan is to combine the power of the two types of analysis, since a preliminary study indicates that the shortcomings of one are the strengths of the other (Section 2).

3.4.4 Giving Specifications

Following the example of Ko's work in [6], specifications could be used to tell our detection tool what the normal behaviour of the target program is. There are three main ways to give a specification:

1. Specify exactly what the application does.
2. Specify what it can and cannot do in general.
3. Specify a suspected vulnerability.

The first choice is impractical for long programs because of the sheer length of the specification, since one must "reverse-specify" the application.

The third choice is much easier to use, but it lacks generality: too much detail about actual time bombs must be provided. Moreover, this approach is useless against new time bombs. This approach suffers from the shortcomings of virus detectors: it is effective only against known attacks.

The authors believe that the second choice is the way to go. In the particular case of time bombs, a specification might be extremely simple: should the application base any of its normal operations on the current time? Yes or no?

Of course, finer grain specifications are needed in the case where an application is required to use the time. The language should be able to specify that a program needs the time for one particular input only, and for no other. In a fully automated tool, the administrator should be able to tell the monitor that "If the user requests that particular action, then the application should be allowed to use the time. Otherwise, it should not." For example, in a virus detection tool, if the user requests a scan every day at 6 o'clock then the monitor should know that it is permissible for the application to check the time against 6 o'clock, and not raise a warning. In any other situation it should raise one.

Specifications could also be useful to organise our knowledge of malicious code. For instance, if a grammar to specify malicious code is defined, a tool could be devised that would not need to be recompiled simply to add new knowledge to it. It could have a separate database that would be checked dynamically.

The two levels of specification could (and probably should) be combined. For example, to simplify specification writing, there should be only one way of specifying "get the time." For example, let the `GetSystemTime` function be the one and only function to get the time in our user-level specification. Then the user could say something very simple like:

```
SYSTEMTIME systemTime;
IF( GetSystemTime(&systemTime) )
   THEN violation();
```

Internally, our monitoring application would look in its database where all the different possibilities of getting the time are specified, link them with the `GetSystemTime` specification, check for them, and raise a violation if any is used.

In the end, the user-level specification might be as simple as a checklist showing all the possible malicious actions our tool can detect. The user would need only to check the kind of malice he wants to be warned against.

3.5 Time Bomb Detection – Conclusion

In this chapter, the process of creating and using a time bomb was examined very closely via the example of the expiration scheme for *SoftBomb*. It has shown that, in assembly language, the process can be quite complex. The instructions required might be spread through a large part of the executable code.

Although in this particular case the limitations imposed on DLL coding forced all the malicious code to be in one function, we will not always be this lucky. In a normal application, the malicious code could be scattered around the entire executable file. For example, an intelligent programmer could do what *SoftBomb* does — change the string `system32` to the string `win.ini` — while remaining unnoticed, by altering one letter at a time in seven different functions. The activity would certainly be more difficult to spot. Only the attacker would know which functions to execute to get the wanted result. He could make the process even more complicated by specifying an order for the function calls. By adding simple checks, he could see to it that the malicious function would be executed only by a precise sequence of operations, in effect creating a trapdoor.

Many ways to get the time and date, or to set timers to execute a task at a particular time have been described. The list may not be exhaustive, but it constitutes a vital first step towards identifying all the possible ways of getting system time.

Several approaches were proposed for a tool to detect time bombs. Although not all have been tested experimentally and no fully working prototypes have been created, the authors feel that the ideas expressed in this chapter could be useful not only toward the detection of time bombs, but also toward the goal of detecting any other kind of malicious code. Of course, any such steps would require that the extensive analysis that was performed for time bombs be extended to other forms of malicious code. The authors think that such a tool could relatively easily be adapted to provide continuous protection, as opposed to being used only in a testing environment. Because many errors in computer systems are the result of user error, such a tool would certainly be valuable.

4 COTS against COTS

Three commercially available products that offer protection against malicious code were examined, concentrating on those that can work at the desktop level — since most COTS will be installed via a CD-ROM or an intranet — and on those that are specifically designed to block malicious code — thus excluding network intrusion detectors. Most of the products examined have sister versions that can work at the network level. Although the selection is by far not exhaustive, most of the other available products have the same basic functionalities. Plus, almost all of these tools work only on mobile code (Java, ActiveX, JavaScript...), with some offering very basic protection against COTS that does not come from the network (e.g. CD-ROM, diskettes). This is the case for two of the three presented.

Neeley [11] gives a more complete list of available products, along with a good overview of what is at stake when dealing with this sort of program. Missing from this list are newer products from companies such as Norton and McAfee. The list of potential products is growing very rapidly, most of them claiming that they are the "First Product to Offer Complete Protection for Web Users". It can be rather confusing to determine exactly what level of protection is provided by current products.

4.1 Classifying

Randall [12] roughly defines three approaches to security for personal PCs. Most products today combine them to offer a wide range of protection. The three are:

Personal Firewall (Blocking) A simple gatekeeper that allows the user to control what passes in and out of communication ports. This only blocks certain channels, without any form of content analysis, and is therefore highly efficient speed-wise. Most firewall vendors have a personal PC version available. eSafe Protect Desktop uses this technology to block communication ports.

Sandbox Popularised by Java, the Sandbox model encloses the application in a virtual environment in which it can cause no harm. eSafe Protect Desktop also uses this technology to prevent selected programs from accessing specifically enumerated resources. This approach appears promising, but "Because of the high potential for programming errors, 'the sandbox is almost a moot point. You can't count on the sandbox for security,' says Ted Julian, a senior analyst for Forrester Research International" [11].

Scanning Much like current virus scanners, the tool scans the mobile code before downloading and executing it to see if it contains potentially malicious actions. It if does, the code if prevented from reaching the system. This technique is quite hard on system performance. Finjan's SurfinShield and Trend Micro's PC-cillin 6 both use this technique.

Let us examine these products in a little more detail and then discuss their shortcomings.

4.2 The Tests

Three products were tested, to give an indication of what is available on the market. The test consisted of trying to run the following documented hostile applets or ActiveX controls:

	Hostile Applets	Tiny Killer App	Exploder	Runner	ActiveX Check	Spy
eSafe Protect Desktop	9/9 blocked	NB	B	NB	13/17 blocked	NB
Surfinshield Online	9/9 blocked	NB	B	B	13/17 blocked	NB
PC-cillin	9/9 blocked	NB	B	NB	13/17 blocked	NB

Table 1: Comparison of what the three products successfully blocked (B: Blocked, NB: Not Blocked)

LaDue's Collection of Increasingly Hostile Applets [7] 9 documented hostile applets.

Tiny Killer App(let) [9] A small applet that forces Netscape to cause an access violation, thereby killing the browser.

McLain's Exploder [10] Exploder is an ActiveX control that performs a clean shutdown of your computer.

McLain's Runner [10] Runner is an ActiveX control that demonstrates how to run an arbitrary program on the browser's machine.

Smith's ActiveX checks [13] Checks for vulnerabilities to 17 documented hostile ActiveX controls.

Tegosoft's Spy [4] An ActiveX control that demonstrates how it can intercept what the user types on his keyboard. When activated, it replaces every key one types in NotePad into the sequence of letters forming `www.tegosoft.com` — press any key, and w appears, press 16 random keys and the whole sequence appears, the next key begins a new line and it starts again.

The Java applets were tested on both Netscape and MS Internet Explorer, while the ActiveX controls work only in MS Internet Explorer.

The results of the tests are presented in Table 1. All the products perform quite well on known and documented mobile code attacks, but unfortunately it is easy to find an attack that defeats them, as indicated by the tiny killer applet that eludes all three products.

Another interesting detail is that Tegosoft's SmartLoader, the ActiveX control responsible for loading the Spy control, was blocked at first by SurfinShield. This is interesting because the control is signed and perfectly legitimate. This illustrates the fact that legitimate software can easily be considered illegitimate. The line is not clear between what is legitimate and what is not.

eSafe Protect Desktop 2.1 According to its advertising, Aladdin Knowledge System's product "is a cutting edge, personal Internet content security solution for individual PC users, at home or at work. eSafe Protect Desktop includes a patent-pending anti-vandal sandbox module, an advanced, ICSA-certified anti-virus scanner, a unique personal firewall module, and a comprehensive resource protection system." (`http://www.esafe.com/products22/products.html`).

It includes an interesting sandbox feature that can, for example, prevent all programs from modifying the desktop, or prevent a specific application from accessing certain directories. It works as a super Access Control Lists (ACL) in the sense that, in addition to normal ACLs functions, which restrict access based on users, it allows access to be restricted for individual programs. Although this feature was of great interest in theory, in reality it did not stop the installation of the annoying WinZip icon on the desktop (☺).

The interface is attractive, although rather complex, as is the case with most tools in this category. This is definitely not entry-level material and, contrary to the publicity, it is not usable by the average user. As is so often the case, the default options do not offer the best level of protection the program can provide, which can be misleading.

The product provides full antivirus protection and it also creates and manages file integrity checks. Overall, it is a good contender and it is worth following up future versions.

Surfinshield Online 4.7 Finjan Software's product "enables companies to conduct e-business safely by providing proactive, run-time monitoring of executables, Java and ActiveX on corporate PCs" (`http://www.finjan.com/products_home.cfm`).

It uses a central server holding security policies and central knowledge. When a desktop detects a security breach, it informs the server, which immediately informs all clients, providing immediate protection for the entire network as soon as a breach occurs. Only the client is provided in the online version, the one tested; the server resides at Finjan's. Although this configuration limits options, it was used to provide a fair comparison with the other products.

A disturbing event occurs during installation: the product says that it is going to "adjust" your browsers. It is easy to understand that such a tool needs to make some changes to a system to protect it effectively. But what exactly does it do? Is the change safe? Does one really want a COTS product to change local programs?

LaDue [8] virulently describes the weaknesses of this product. In summary, he says that SurfinShield is only good at providing protection against known attacks. Even then, it is not very good since the "knowledge" is based on a list of URLs. LaDue's article is a bit dated and probably too rash — the product has definitely improved since the time of the judgement. But his drastic comments are indicative of shortcomings of all products currently on the market. Many of the general inadequacies common to most of these security products are discussed in the next subsection.

The product does not have antivirus protection; a separate tool is needed.

An interesting feature — once again, at least on paper, — is the SafeZone, which monitors the execution of a binary program. It is launched automatically on programs that come from the net and it can be launched manually to monitor a specific program. It stops a program from reading or writing files, making network connections, writing to the registry, or starting other programs. This works fine except that, frankly, what useful programs can one run under such constraints? This example illustrates a key concept in security: usability versus security.

PC-cillin 6.07 Trend Micro advertises this product as "all the protection you need to face the new Internet frontier!" (`http://www.antivirus.com/pc-cillin/products.htm`).

It is a typical example of new, emerging products. It is primarily an antivirus program that doubles as a malicious mobile code detector. As users become more aware of the security problems inherent to Internet use, they realise they need some form of protection. Companies see this opportunity and jump on it by offering their own products.

PC-cillin looks like a pretty good antivirus product — no tests were made of that use — but it is certainly lacking as a personal protection tool from the hazards of the Internet. Its single primary interface scans only incoming mobile code, much the same way that an antivirus program does. There are no facilities to protect from malicious files from CDs or an intranet — unless, of course, they contain viruses.

4.3 Shortcomings of COTS Desktop Security Products

First, because they are based on a priori knowledge of malicious code, they are unable to deal with unknown attacks. This is clearly not an acceptable approach since attackers will always be a step ahead of security tools. Furthermore, because commercial products of this nature are often rushed to delivery, they are quite error-prone. The problem is similar to that of current antivirus utilities, but more serious. It would be a full-time job for many users just to keep up with the patches and, given that in most cases the list of attacks must be updated manually, it is easy to understand that this is not a promising long-term solution. Future tools need to be able to detect suspicious behaviour on their own. Some form of "intelligence" is needed.

Second, they are usually quite complicated to use. Even though the actual level of customisation is rather limited, an expert is required most of the time, just to keep the product running without overpowering the routine activities of the system's users. Future tools need a powerful specification language for expert users and a very simple interface for everyday users. Then security administrators can set very precise policies and average users can be successfully protected without being annoyed by repetitive and often unspecific alert messages.

Along the same train of thought, the more tools one has or needs, the more confusion will be brought to the average user. For example, antivirus protection is a must for an organisation, as is protection from malicious code and intrusion. A perfect security tool would incorporate protection against all of these aspects in one package, providing the user with a single, consistent interface for all aspect of security.

Finally, most of the COTS Internet security products do not even attempt to address the problem of security in COTS obtained in executable format (e.g. MS Office, Eudora, MapObjects, and so on), which probably still account for the vast majority of purchased COTS. A complete tool obviously needs to be able to address the problems of binary programs.

5 Conclusion

Dynamic detection of malicious code has been outlined in this paper. This is one of the best techniques to detect malicious activity since it acts at the lowest possible level: processor instructions. Thus the MaliCOTS team concentrates its research effort on collaborative techniques that include both static and dynamic tools. It is our hope that dynamic analysis can complement static analysis and overcome its shortcomings. This will ensure the rigorous and efficient integration of COTS packages even when the source code is not available.

One of our top priorities at this time is to formalise the expression of security policy using a good specification language to discriminate malicious activities from acceptable behaviours. This requires very fine granularity. Currently, various design possibilities for a common security specification language are being examined within our research effort and a technology watch monitors commercial solutions.

Our team welcomes international collaboration.

References

[1] R. Charpentier and M. Salois. MaliCOTS:Detecting Malicious Code in COTS Sofware. In *Commercial Off-The-Shelf Products in Defence Applications "The Ruthless Pursuit of COTS"*, Neuilly-sur-Seine Cedex, France, Apr. 2000. NATO, RTO.

[2] S. Cho. Win32 Disassembler. `http://www.geocities.com/SiliconValley/Foothills/4078/`, Oct. 1998.

[3] C. Colby. *Semantics-based Program Analysis via Symbolic Composition of Transfer Relations*. PhD thesis, Carnegie Mellon University, Aug. 1996.

[4] T. Inc. Samples. http://www.tego.com/WebFrameSets/OcxControlKit/Samples.htm, 2000.

[5] C. L. Jeffery. A Framework for Monitoring Program Execution. Technical Report 93-21, University of Arizona, July 1993. Department of Computer Science, http://ringer.cs.utsa.edu/research/alamo/.

[6] C. C. W. Ko. *Execution Monitoring of Security-Critical Programs in a Distributed System : A Specification Based Approach.* PhD thesis, University of California Davis, Aug. 1996. Graduate Division.

[7] M. D. LaDue. A Collection of Increasingly Hostile Applets. http://www.rstcorp.com/hostile-applets/index.html.

[8] M. D. LaDue. The Rube Goldberg Approach to Java Security. http://www.rstcorp.com/hostile-applets/rube.html, 1998.

[9] G. McGraw and E. Felten. Java Security Hotlist. http://www.rstcorp.com/javasecurity/hotlist.html.

[10] F. McLain. ActiveX or How to Put Nuclear Bombs in Web Pages. http://www.halcyon.com/mclain/ActiveX/, 1997.

[11] D. Neeley. How to Keep Out Bad Characters. Security Management Online, 1998. http://www.securitymanagement.com/library/000599.html.

[12] N. Randall. Personal Security Suites. http://www8.zdnet.com/pcmag/features/personal_security/_open.htm, 1997.

[13] R. M. Smith. ActiveX Security Check Page. http://www.tiac.net/users/smiths/acctroj/axcheck.htm, 1999.